

Shared Memory Parallelization

OpenMP



Thomas J. Watson Research
Center
PO Box 218
Yorktown Heights, NY 10598



Outline

- Local Information
- What is Shared Memory Parallelization
- Variable Scoping
- Work Sharing
- Performance Issues



What is Shared ("Symmetric") Memory Parallelization

- All processors can access all the memory in the parallel system (node)
 - The time to access the memory may not be equal for all processors - not necessarily a flat memory
- Parallelizing on a SMP does not reduce CPU time - it reduces wallclock time
- Parallel execution is achieved by generating threads which execute in parallel
 - Number of threads is (\sim) independent of the number of processors



What is Shared Memory Parallelization (*continued*)

- Overhead for SMP parallelization is large
 - 10's of microseconds
 - Size of parallel work construct must be significant enough to overcome overhead
- Runtime handling of parallel threads important
- SMP parallelization is degraded by other processes on the node - important to be dedicated on the SMP node
- Remember Amdahl's Law - Only get a speedup on code that is parallelized



Variable Scoping

- Most difficult part of Shared Memory Parallelization
 - What memory is Shared
 - What memory is Private - each processor has its own copy
- Fortran concept of Memory
 - Global
 - Shared by all routines
 - Local
 - Private to routine



Variable Scoping Rules

- Private Variables
 - Thread scope only
 - Scalar variable that is set and then used within the DO
 - An array whose subscript is constant with respect to the PARALLEL DO and is set and then used within the DO
- Shared Variables
 - Everything Else
- Shared is the default (for ALL variables)



Fortran vs SMP Scoping

- Whenever a Fortran GLOBAL variable is scoped PRIVATE or when a Fortran LOCAL variable is scoped SHARED problems arise
 - Variable passed into a routine scoped private - FIRST Value getting and LAST value setting
 - COMMON block variable within a called routine needs to be scoped private



OpenMP Directives

- <http://www.openmp.org>
- Comment line directives for
 - Scoping Data
 - Specifying Work Load
 - Synchronization of threads
- Function calls for obtaining information about threads



OpenMP Directives

- Scoping Variables
 - Default is **SHARED**
 - Can be set to **NONE** or **PRIVATE**
- Nothing like AUTOSCOPE - user responsible for scoping anything that is contrary to default
- Scoping cannot be done within a subroutine called from the parallel DO loop - except with **THREADPRIVATE**



OpenMP Directives

- `!$OMP PARALLEL / !$OMP END PARALLEL`
 - "Fork" and "join"
 - Indicate a parallel region for each thread to execute - must scope all variables within region



OpenMP Directives

- `!$OMP PARALLEL DO / !$OMP END PARALLEL DO`
- Fork and join, plus work sharing
 - Indicate a parallel do for all threads to shared in work - must scope all variables within region - Can specify Worksharing



OpenMP Directives

- !\$OMP DO / !\$OMP END DO
- Work sharing (within a parallel region)
- Note the optional "END DO"
- Indicate a parallel do for all thread to shared in work - May Scope variables
- Can specify Worksharing



Variable Scoping

```
read *,n
sum = 0.0
call random (b)
call random (c)
!$OMP PARALLEL DO
!$OMP&PRIVATE (i)
!$OMP&SHARED (a,b,n)
!$OMP&REDUCTION (+:sum)
do i=1,n
a(i) = sqrt(b(i)**2+c(i)**2)
sum = sum + a(i)
Enddo
!$OMP PARALLEL ENDDO
end
```

Reduction: Note that
reduction variable
cannot be an array!



Variable Scoping

```
read *,n
sum = 0.0
call random (b)
call random (c)
!$OMP PARALLEL
!$OMP PRIVATE (i,sump)
!$OMP SHARED (a,b,n,c,sum)
    sump = 0.0
!$OMP DO
    do i=1,n
        a(i) = sqrt(b(i)**2+c(i)**2)
        sump = sump + a(i)
    enddo
!$OMP CRITICAL
    sum = sum + sump
!$OMP ENDCRITICAL
!$OMP END PARALLEL
end
```

Another ("Traditional") way
of performing a reduction



Variable Scoping

```
subroutine example4(n,m,a,b,c)
  real*8 a(100,100),B(100,100),c(100)
  integer n,i
  real*8 sum
!$OMP PARALLEL DO
!$OMP PRIVATE (j,i,c)
!$OMP SHARED (a,b,m,n)
  do j=1,m
    do i=2,n-1
      c(i) = sqrt(1.0+b(i,j)**2)
    enddo
    do i=1,n
      a(i,j) = sqrt(b(i,j)**2+c(i)**2)
    enddo
  enddo
end
```

Each processor needs
a separate copy of *j,i,c*
everything else is
Shared

What about *c*?
c(1) and *c(n)*?



Variable Scoping

```
subroutine example4(n,m,a,b,c)
  real*8 a(100,100),B(100,100),c(100)
  integer n,i
  real*8 sum
!$OMP PARALLEL DO
!$OMP PRIVATE (j,i)
!$OMP SHARED (a,b,m,n)
!$OMP FIRSTPRIVATE (c)
  do j=1,m
    do i=2,n-1
      c(i) = sqrt(1.0+b(i,j)**2)
    enddo
    do i=1,n
      a(i,j) = sqrt(b(i,j)**2+c(i)**2)
    enddo
  enddo
end
```

Need First Value of *c*

Master copies it's
c array to all threads
prior to DO loop

FIRSTPRIVATE entails data
motion (copying).



Variable Scoping

```
subroutine example5(n,m,a,b,c)
  real*8a(100,100),B(100,100),c(100)
  real*8 cc(100)
  integer m,n,i
  real*8 sum
!$OMP PARALLEL
!$OMP PRIVATE (j,i,cc)
!$OMP SHARED (a,b,m,n)
  cc(1) = c(1)
  cc(n) = c(n)
!$OMP DO
  do j=1,m
    do i=2,n-1
      cc(i) = sqrt(1.0+b(i,j)**2)
    enddo
    do i=1,n
      a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
    enddo
  enddo
!$OMP END DO
!$OMP END PARALLEL
end
```

Need First Value of *c*

User copies what part of
c is needed to all threads
prior to DO loop



Variable Scoping


```
!$OMP PARALLEL
!$OMP PRIVATE (j,i)

!$OMP SHARED (a,b,m,n)
  cc(1) = c(1)
  cc(n) = c(n)
!$OMP DO
  do j=1,m
    do i=2,n-1
      cc(i) = sqrt(1.0+b(i,j)**2)
    enddo
    do i=1,n
      a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
    enddo
  enddo
!$OMP END DO
  if(j.eq.m+1)then
    do i=1,n
      c(i) = cc(i)
    enddo
  endif
!$OMP END PARALLEL
```

What about last value
of c?

Each thread performs
final copy.

Could use "LAST
PRIVATE"



Calling an External from a Parallel Loop

```
subroutine example5(n,m,a,b,c)
  real*8 a(100,100),B(100,100),c(100)
  integer m,n
!$OMP PARALLEL DO
!$OMP PRIVATE (j)
!$OMP SHARED (a,b,m,n)
  do j=1,m
    call doit(j,n,a,b)
  enddo
end
```

```
subroutine doit(j,n,a,b)
  real*8 a(100,100),B(100,100)
  COMMON cc(100)
  do i=2,n-1
    IF(a(i,j).gt.SIN(b(i,j)))THEN
      cc(i) = sqrt(1.0+b(i,j)**2)
    ENDIF
  enddo
  do i=1,n
    a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
  enddo
end
```



Calling an External from a Parallel Loop

```
subroutine example5(n,m,a,b,c)
  real*8 a(100,100),B(100,100),c(100)
  integer m,n
!$OMP PARALLEL DO
!$OMP PRIVATE (j)
!$OMP SHARED (a,b,m,n)
  do j=1,m
    call doit(j,n,a,b)
  enddo
end
```

```
subroutine doit(j,n,a,b)
  real*8 a(100,100),B(100,100)
!$OMP THREADPRIVATE (/BCOM/)
COMMON/BCOM/ cc(100)
  do i=2,n-1
    IF(a(i,j).gt.SIN(b(i,j)))THEN
      cc(i) = sqrt(1.0+b(i,j)**2)
    ENDIF
  enddo
  do i=1,n
    a(i,j) = sqrt(b(i,j)**2+cc(i)**2)
  enddo
end
```

Blank Common cannot appear on **THREADPRIVATE**
How about first value setting???

Not in xlf Version 6.1

... Called **"threadcommon"**



Work Sharing Directives

- SCHEDULE (type,n)
 - Runtime
 - Scheduling is controlled by runtime environment variable
 - OMP_SCHEDULE (Not in xlf 6.1)
 - XLSMPOPTS on xlf Version 6.1
 - (Static,n)
 - Iterations are divided into chunks and pieces are statically assigned to threads in a round-robin fashion (Default n is iteration count/parthds)



Work Sharing Directives

■ SCHEDULE

▸ (Dynamic,n)

- Work is divided into chunks of size n . As each thread finishes a chunk it dynamically obtains the next set of iterations. (default of n is 1)

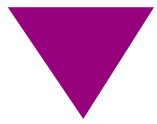
▸ (Guided,n)

- Dynamic with chunksize starting at iterations/parthds, then exponentially decreasing to n . (default of n is 1)



Tradeoff Load Balancing and Reduced Overhead

- The larger the size (GRANULARITY) of the piece of work, the lower the overall thread overhead.
- The smaller the size (GRANULARITY) of the piece of work, the better the dynamically scheduled load balancing



OpenMP for C

- Specification 1.0, October 1998
- Same functionality as OpenMP for FORTRAN
- Differences in syntax:
 - `#pragma omp parallel`
 - `#pragma omp for`
- Differences in variable scoping:
 - variables "visible" when `#pragma omp parallel` encountered are shared by default
 - static variables declared within a parallel region are also shared
 - heap allocated memory (malloc) is shared (but pointer can be private)
 - automatic storage declared within a parallel region is private (ie, on the stack)



What About Automatic?

- xlf has a very good automatic parallelizer that might do a good job on a User's program.
 - When applying it across an entire program, some loops may slow down, some may speed up - you should be prepared to time individual loops before and after and then selectively parallelize what you want

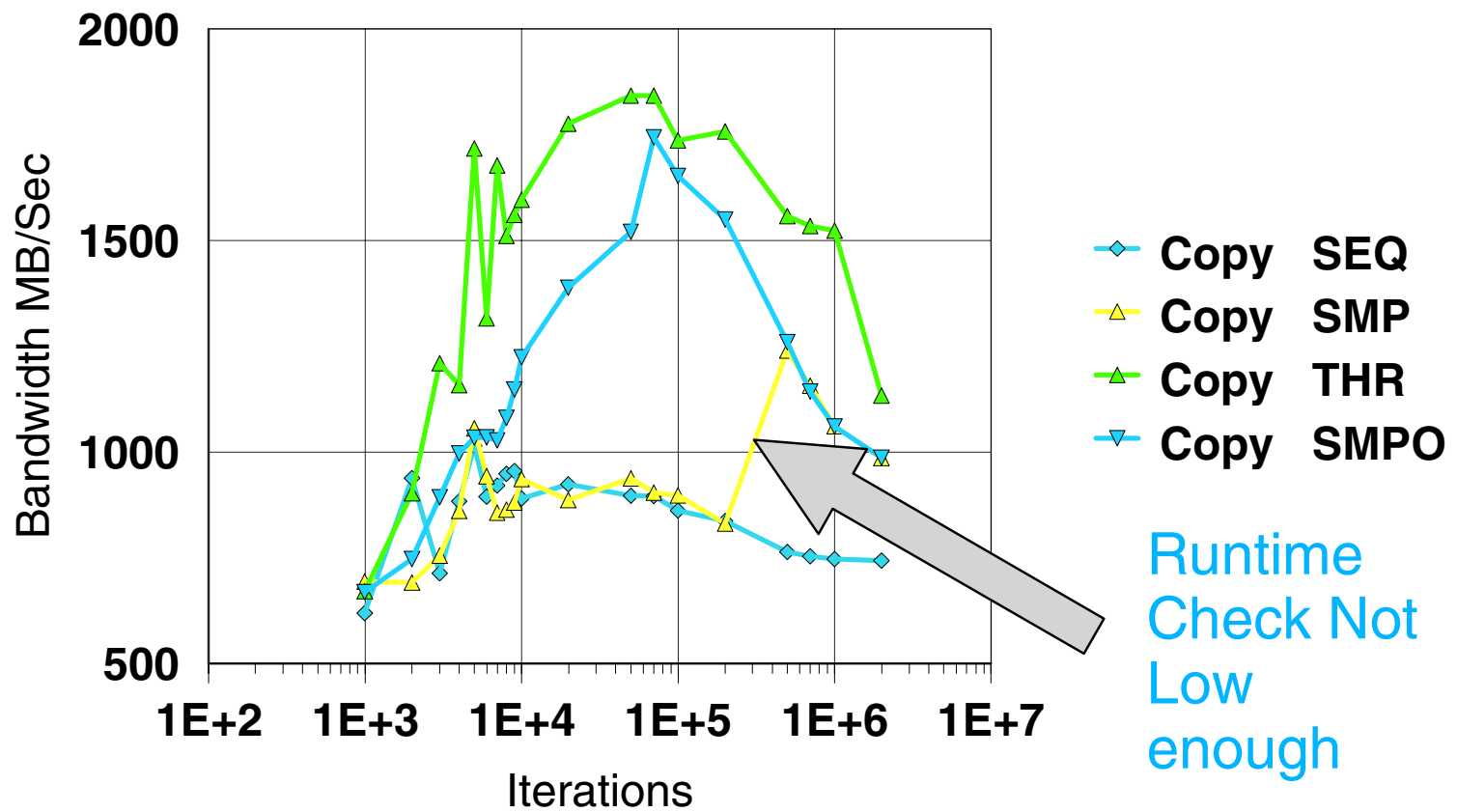


What About Automatic?

- xlf has a very good automatic parallelizer that might do a good job on a User's program.
 - Runtime checking of overhead controlled by runtime environment variable

Winterhawk 2-Processor

Stream Rates for Scale





Runtime Enviroment Vars

- Some will probably change with Version 7.1 - OpenMP standard
 - XLSMPOPTS
 - parthreshold = num
 - ◆ specifies time in milliseconds below which the loop will run in serial
 - seqthreshold = num
 - ◆ specifies time beyond which previous sequential loop will be run in parallel
 - profilefreq=num
 - ◆ frequency with which loop should be analyzed
 - ◆ = 0 All profiling turned off



Runtime Management of Threads

- When the system encounters the first Parallelized DO loop the Master generates the worker threads and begins working on a chunk of the Parallel DO loop
- After the first Parallel DO loop is executed, all the worker threads are put to sleep - regardless of the spin Environment variable



Runtime Management of Threads ***(continued)***

- When the next DO loop is encountered, the Master wakes up a first worker thread, the Master continues to work on the parallel loop. The first worker thread wakes up the second worker thread and starts to work on the parallel loop. The Master may have already started a second piece of work. The second thread wakes up the third,



Runtime Enviroment Vars

- Some will probably change with Version 7.1 - OpenMP standard
 - XLSMPOPTS
 - parthds=num
 - ◆ default - number of processors
 - stack=num
 - ◆ default - 4194304
 - spins=num (Only for locks)
 - ◆ default - 100
 - yields=num (Only for locks)
 - ◆ default - 10



Runtime Management of Threads

- Supply a runtime environment variable to specify that the threads should be put in a spin state rather than put to sleep.
- **Environment variable:**
SPINLOOPTIME=5000
 - "spinwait"
 - This will saturate the CPU, not good if the node is timeshared
 - This will effectively reduce the overhead of threads joining the work section



Summary

- Full OpenMP 1.0:
 - C for AIX 5.0
 - November, 1999
- XLF 7.1
 - March, 2000
- Synchronization and scheduling are comparable to SGI and Sun
- More concern with load balance and locality
- USE new libxlsmp.a
- Old version has poor performance